Problem Set 4
data structures and algorithms 2 - fall 2023

vgs3qt
due november 3, 2023 at 11:59 pm

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: Yining Liu(yl2nr), Yuhao Niu(bhb9ba)

Sources: list your sources

problem 1 *Dynamic Programming*

1. If a problem can be defined recursively but its subproblems do not overlap and are not repeated, then is dynamic programming a good design strategy for this problem? If not, is there another design strategy that might be better?

   Solution:

   No. When the subproblem doesn't repeat, it is useless to save them into memory because we never read them and thus it does not reduce running time. An alternative strategy that fits this situation more is the Divide and Conquer, Greedy Algorithm, or other straightforward recursion or iterative methods

2. As part of our process for creating a dynamic programming solution, we searched for a good order for solving the subproblems. Briefly (and intuitively) describe the difference between a top-down and bottom-up approach.

   Solution:

   The pop-down approach is we start with the original (high-level) problem and break it down into smaller subproblems recursively. We solve each subproblem as needed but avoid redundant work by memoizing (caching) the solutions to subproblems. We check if we solved it before by checking our memory. If not solved we stored the result into a memory.

   The bottom-up approach is that we start with the smallest subproblems (base cases) and iteratively build up solutions for larger subproblems. Solves each subproblem only once and stores its solution in memory.

   They also differ in their execution order and the way they organize and retrieve subproblem solutions.

PROBLEM 2 *Birthday Prank*

Prof Hott's brother-in-law loves pranks, and in the past he's played the nested-present-boxes prank. I want to repeat this prank on his birthday this year by putting his tiny gift in a bunch of progressively larger boxes, so that when he opens the large box there's a smaller box inside, which contains a smaller box, etc., until he's finally gotten to the tiny gift inside. The problem is that I have a set of n boxes after our recent move and I need to find the best way to nest them inside of each other. Write a **dynamic programming** algorithm which, given a $fits(b_i, b_j)$ function that determines if box b_i fits inside box b_j, returns the maximum number of boxes I can nest (i.e. gives the count of the maximum number of boxes my brother-in-law must open).

Solution:

1. We use iterative (bottom-up) dynamic programming because we don't know the order of size of those boxes so we start from any one box. Our function starts from any box b_i and returns a value *numbox*. We iteratively find the number of boxes a box can nested for every box and return the maximum *maxbox*. For any box b_i, we check if it fits in box b_j and recursively find the *numbox* of all boxes that can nest in b_i, selecting the maximum number and plus 1 (b_i itself), updating the value in list *dp* to record the total number of boxes *numbox* b_i can nest. List *dp* is a list that stores the *numbox* each box can nest. After looping all n boxes, we return the maximum *numbox* from the list *dp* as the answer.

2. We create a list *dp* of size n to store the maximum number of nested boxes ending with each box. Initialize $dp[i]$ to None for all i from 0 to $n-1$. For each box b_x, our function *CheckNumNested(b_x)* first checks if the $dp[b_x]$ is None. If it is not, it means we have already finished the recursive function before and just return the value in $dp[b_x]$.

If $dp[b_x]$ is None, we first make $numbox_x = 1$ (it at least has itself) and loop through all boxes. For every other box b_y, we first check $fits(b_x, b_y)$ to see if it is inside b_y.

If it fits, we check if $dp[b_y]$ is None. If not None, we read the data as $numbox_y$. If None, $numbox_y$ = *CheckNumNested(b_y)*. Use the function to find it's *numbox*.

We finally compare $1 + numbox_y$ with $numbox_x$. If the former part is greater, we update $numbox_x$ to be $1 + numbox_y$. Otherwise, we do nothing because we want maximum. After looping through all n boxes for b_x, we update $dp[b_x]$ to be $numbox_x$. The base case occurs in the smallest box that only has itself, so return the $numbox = 1$.

3. We repeat step 2 for all n boxes and stop when n values in *dp* is not None. We finally return the highest value in *dp*, which is the maximum number of boxes we can nest.

PROBLEM 3 *Arithmetic Optimization*

You are given an arithmetic expression containing n integers and the only operations are additions ($+$) and subtractions ($-$). There are no parenthesis in the expression. For example, the expression might be: $1+2-3-4-5+6$.

You can change the value of the expression by choosing the best order of operations:

$$((((1+2)-3)-4)-5)+6 = -3$$
$$(((1+2)-3)-4)-(5+6) = -15$$
$$((1+2)-((3-4)-5))+6 = 15$$

Give a **dynamic programming** algorithm that computes the maximum possible value of the expression. You may assume that the input consists of two arrays: `nums` which is the list of n integers and `ops` which is the list of operations (each entry in `ops` is either `'+'` or `'-'`), where `ops[0]` is the operation between `nums[0]` and `nums[1]`. *Hint: consider a similar strategy to our algorithm for matrix chaining.*

Solution:

We will use the top-down strategy for this question. From the top-level problem (the overall equation), we recursively calculate the maximized outcome by finding the maximum result of different possible combinations. We split the top-level problem into smaller sub-problems.

1. Let the recursive function be *MaxOutcome()*, and the memory matrix be *dp*. We start from the top level, the whole equation. Thus, we call the function: *MaxOutcome(nums, ops, start, end)*, where the *nums* is now the entire number-array, *ops* is the entire operation-array, the *start* is 0, and the *end* is n. We assume there is a total of n numbers. Here, we have n different ways to split the array, and we split the top-level problem into smaller problems by splitting them into two parts and calculating the maximum combinations.

k is an index that goes from start (0) to end (n). For the split: *MaxOutcome(nums, ops, start* $= 0$, *end* $= n$*)*
$=$ *max* (*MaxValue*, *MaxOutcome(nums, ops, start* $= 0$, *end* $= k$*)* $+/-$ *MaxOutcome(nums, ops, start* $= k+1$, *end* $= n$*)*).

$+/-$ is determined by *ops[k]*. When *start* $=$ *end*, return *nums[start]*; *MaxValue* is the valued stored in *dp[start][end]*. It is 0 before it is changed by the recursive call.

2. We continue the recursive steps till we reach the base case when the *start* $=$ *end* and there is only one number so we return that number. Then, the function will add or subtract the two numbers from the returned and trace backward.

3. Each step will store the max(...) as *MaxValue* in *dp[start][end]*. Finally we returned to the top-level problem and got all answers to the sub-problem in *dp*. Now we return the result of *MaxOutcome(nums, ops, 0, n)*, which is *dp[0][n]*.

PROBLEM 4 *Stranger Things*

The town of Hawkins, Indiana is being overrun by interdimensional beings called Demogorgons. The Hawkins lab has developed a Demogorgon Defense Device (DDD) to help protect the town. The DDD continuously monitors the inter-dimensional ether to perfectly predict all future Demogorgon invasions.

The DDD allows Hawkins to predict that i days from now a_i Demogorgons will attack. The DDD has a laser gun that is able to eliminate Demogorgons, but the device takes a lot of time to charge. In general, charging the laser for j days will allow it to eliminate d_j Demogorgons.

Example: Suppose $(a_1, a_2, a_3, a_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to fire the laser at times 3, 4 in order to eliminate 5 Demogorgons.

1. Construct an instance of the problem on which the following "greedy" algorithm returns the wrong answer:

   BADLASER$((a_1, a_2, a_3, \ldots, a_n), (d_1, d_2, d_3, , \ldots, d_n))$:
   Compute the smallest j such that $d_j \geq a_n$, Set $j = n$ if no such j exists
   Shoot the laser at time n
   if $n > j$ then BADLASER$((a_1, \ldots, a_{n-j}), (d1, \ldots, d_{n-j}))$

   Intuitively, the algorithm figures out how many days (j) are needed to kill all the Demogorgons in the last time slot. It shoots during that last time slot, and then accounts for the j days required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

   Solution:

   Lets assume $(a_1, a_2, a_3, a_4) = (1, 1, 1, 6)$ and $(d_1, d_2, d_3, d_4) = (1, 3, 5, 7)$. Based on the greedy algorithm, it will produce an answer that we shoot on all four days to eliminate $1 + 1 + 1 + 1 = 4$ Demogorgons because it fits $d_j \geq a_n$. However, if we hold the laser and shoot on day4 only, it can eliminate 6 Demogorgons.

2. Given an array holding a_i and d_j, devise a dynamic programming algorithm that eliminates the maximum number of Demogorgons. Analyze the running time of your solution. *Hint: it is always optimal to fire during the last time slot.*

   Solution:

   We use top-down dynamic programming to solve this problem recursively. We calculate and store the number of Demogorgons killed by the gun that charges k days. k can equal to total days n. Let the function be *MaxKilled(Days)*. Let the memory dictionary be *dp*, setting the initial value to be None. Let the strategy list be *KeepTrack*, initiated with *[n]*, the last shooting day.

   Then we go from the highest-level problem which is day $= n$. We consider how long to charge for the last shot. There are n different ways to split it into two sub-problems, each sub-problem is a lower-level recursion optimal sub-problem. When *Charge Days* $= k$, then the optimal solution will be $min(d_k, a_n) + MaxKilled(Days = n - k)$.

Therefore, we try all k to get the maximum. $MaxKilled(Days = n) = max(d_k, a_n) + MaxKilled(Days = n - k), ...)$. It checks the max value for all k from 0 to n. Finding the k that corresponds to the max value, we add $n - k$ as the earlier shot day to *KeepTrack* to keep track of the days we shot.

We then recursively solve each sub-problem with charged days i. If *dp[i]* is not None, we return the value stored. If it is None, we call $MaxKilled(Days = i) = max(d_j, a_i) + MaxKilled(Days = i - j), ...)$ for all $0 \leq j \leq i$.

We then update *dp[i]* to $MaxKilled(Days = i)$. The base cases are Charged days $= 0$ and 1.

$MaxKilled(Days = 0) = 0$.

$MaxKilled(Days = 1) = min(d_1, a_1)$.

After calculating all cases, the final maximum number of Demogorgons killed is *dp[n]*, the top-level problem. The days we shot is recorded in list *KeepTrack*.

The time complexity is $O(n^2)$, because to iterate all cases we need a time complexity of n, and in each sub-problem, there are n types of combination to solve.